Electronic Notes in Theoretical Computer Science 65 N° 4 (2002)
URL:http://www.elsevier.nl/locate/entcs/volume65.html 13 pages

Extending The Unified Process With Composition

Valérie Monfort (*) Hubert Kadima (**)

(*) Q-Labs
28, villa Baudran
94742 Arcueil cedex
monfort@objectif.fr

(**) Bouygues Telecom
52, avenue de l'Europe
78944 Vélizy Cedex
hkadima@bouyguestelecom.fr

Abstract :

The Unified Process (UP) is an OMG standard. This process is driven by UML use cases and architecture. UP proposes some disciplines but not enough to cope all the needs of the enterprises. One of these needs is the composition. In this paper, we shortly present UP and principles of our methodology named Extended Unified Process (EUP). EUP adds to UP a specific architecture discipline including a process for composition. This paper is an overview of this process.

1 Introduction

There are no known methodology dedicated to composition. Some existing architectural methodologies such as TOGAF [15], *Architecture-Based Development* ABD and PORE [16] integrate partially some composition aspects.

In this paper, we extend the *Object Management Group* (OMG) *Unified Process (UP)* to define a global process integrating some composition tasks as architecture activities. After pointing out the lacks of UP, we present our methodology named *Extended Unified Process* (EUP) as an instance of UP. EUP includes some additional disciplines as architecture. We focus in this paper on the composition process and its integration aspects in a complete EUP process.

2 Extended Unified Process : A Global Process

2.1. The Unified Process

In this section we present main concepts of the *Unified Process* [2] and our extensions.

2.1.1. The concepts

UP is a methodology providing generic recommendations that can be instantiated for different kinds of projects. It provides some guidelines to manage a project by limiting risks. UP includes both the *project* by itself with its organization and the *product* with its different maturity levels and its versions.

The project and its organization allow to split the project into several steps (or *workflows,* or *disciplines*), as requirement management, analysis, design, implementation, test, deployment. All these steps are roughly textually described. Each discipline is expressed by *activities* (tasks to be done), *workers* (people working on the project), *artifacts* (all kinds of things to be delivered : documents, files, code, patterns …).

From maturity level point of view, product development consists of the inception phase (project initialisation), the elaboration phase (designing of architecture, functions, …), the construction phase (at the end of this phase, the product is mature enough to satisfy the client needs) and the transition phase (maintenance of the product).

UP process divides a project into several *iterations* covering all steps of a project. Product life cycle is divided into *phases* (maturity level of the product, or version) that are at their turn divided into iterations. With each iteration, new functions or architecture concepts for instance are added during new increments. This process is driven by UML[1][3] use cases and architecture models and integrate the *risks management*.

2.1.2. The lacks of UP

UP is a generic process which can be applied to all kind of projects ; that require the UP process to be instantiated for each organisation. This is a heavy cost and time consuming task.

Actually, UP provides no solution to the problems of :

- roadmap, risks and project management
- architecture, cartography and composition of processes
- change management process environment process with tools, methodology, quality, configuration management …
- management of the different organisations working on the same project :
- communication between people working together in a project …

2.2 The Extended Unified Process

The *Extended Unified Process* (EUP) is an instance of UP [9][11][12]. It is based on the same semantic, disciplines and *workflows*. It extends UP with additional disciplines improving actual UP missing.

EUP consists of the following disciplines :
- *Elicitation* : It may be necessary to study an information system to draw its *cartography*, collecting needs, criticising existing information system in order to provide *ad hoc* technical solutions … or just to know and to master existing information system. The persons who already did this kind of study know that it may be very tricky. So, this study may require specific techniques to elicit knowledge. An approach is proposed to elicit knowledge and to provide a first draft of business (as business generic invariant components). This approach gathers the best parts of existing and approved methodology as KOD [18] and KADS [19] .
- *Needs and requirements* : Very often, it is difficult to collect needs and to be sure they are correctly and completely expressed. The stabilisation of the needs and their study require a specific process. Prototype is sometimes necessary to be sure to fit to the needs. Sometimes a specific organisation for example to much more involve the client, is necessary. The requirements refine the needs expression with functional and technical dimensions
- *Analysis* takes requirements into account and starts designing classes and first UML models.
- *General and detailed designs* refine the analysis models and use implementation concepts in the modelling.
- *Implementation* leads to code development and unit test classes as results of the modelling and check clients requirements.
- *Integration and its tests* allow to gather several components and to check it works. The integration may be divided into steps or not. If there are steps it may be possible to deliver sources to privileged clients to test the product.
- Then, the product is delivered to the client as *deployment* and tested.
- *Methodology* and *Quality* evaluation, impact the whole development process and needs to be formalised as a specific discipline.
- *Strategy* is a way to think about what to do before starting projects, how to organize them, to define budgets, to design general architecture, to define enterprise methodology, to define a general schedule organizing the whole projects…
- *Project management* allows to manage the risks linked to a project. It means that many parameters have to be taken into account as : resources, needs stabilisation, skills, turnover, architecture, technology mastering …
- *Configuration management* allows to manage configurations of documents, sources, models … Tools support this approach.
- *Tools and environment* impact the project process and are closely tied to the disciplines.
- The *cartography* is a way to capitalize and store enterprise knowledge by designing it with UML models. It is a way to collect models from project process, to gather them and to manage them. Project process feeds by updating the cartography repository which may be a configuration management tool.
- *Architecture* impacts some steps of the project process and is quite necessary to design. In this discipline, we may apply **a top down process** allowing to develop a

component from scratch, **a bottom up process** to compose different components as COTS or a hybrid process including both top down and bottom up approaches when some components are reused and other developed.

All these needs have to be taken into account in methodology and to be expressed as disciplines in EUP. A complete definition of these disciplines in UP is based on distinction between project steps with *core supports* as Project management, Tools and Environment … and *phases description* as Analysis, Design ...

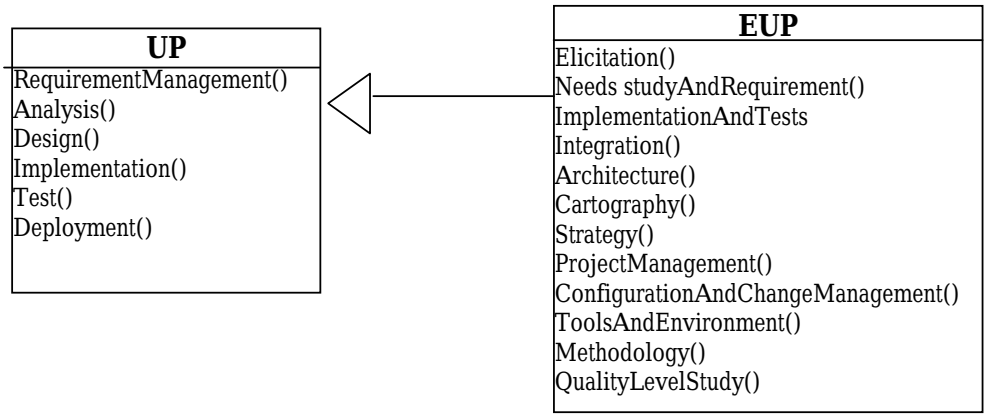

Fig. 1. : Inheritance between EUP and UP

2.3 Risks Management

UP in its project management discipline does not provide a clear approach of the risks management. It is the reason why we have added the risks management aspects in EUP inspired from *Project Management Body of Knowledge* PMBOK [20]. PMBOK proposes some different topics about project risks management, as :

- risks management planning
- risks identification
- qualitative risk analysis
- quantitative risk analysis
- risks response planning
- risk monitoring and control

The risks identification has to be achieved early in the life cycle during the first iterations mainly in inception phase. During the inception phase prototypes are launched. It is a way to evaluate technical, quality and performance risks as soon as possible. Some components may be integrated as a technical proof of concepts and tested. Risks increase when reusing external components. It is very hard to estimate the quality of a component. Moreover, during dynamic composition, external components may be added and combined very easily. As the quality of the components is not proved, it seems very difficult to be sure that the offered services are those expected by the programmer.

An instance of the enterprise methodology is defined with specific guidelines related to the project. This instance is improved iteration after iteration and after several audits. It allows to have a best idea of the kinds of problems that might be met during the project, to search solutions, to train the project team and to search specific skills to join the team.

Planning may include these risks and potential solutions. Very often, project leader uses several scenarios to prevent these risks. For each scenario, costs associated to a schedule and resources, are compared to the budget and justified during the project launching meeting. An other risk about project management is the lack of communication inside the enterprise. For instance, test and integration teams have to receive a schedule of the project very early to add the integration and test tasks in their own schedule..

Sometimes, the organisation of the enterprise adds delay to the schedule, because the aims of the department are not the same or the strategy is not clear. This delay depends on the reactivity of the enterprise to change its organisation.

3 A Component-Driven Approach

Component-based development (CBD) [4][5][7][8][10] is no different from traditional or object-oriented development in that it too needs a process – a process by which the particular characteristics of components can be fully exploited. That process is the framework that the creation of the software solution occurs, taking into account a wide variety of issues, including (but not exclusively) project management, individual skills, tool availability, quality criteria imposed by the end user/client of the software and, of course, reuses strategies.

Component-based development is based on components assembling. It permits static and dynamic composition of components. It is characterized by the following technical issues :

- Clearly component's interface specification . The interface that a component provides must be clearly defined. The same is true for the interfaces it expects of any other component plugged into it.
- Clearly specification of the input and output of the components. A component is usually described using properties, methods and events. Events are the crucial piece added by the component over classical OO classes, to enable the composition of component easily. To plug software parts together, we need a way to describe the equivalent of inputs and outputs for each component, at the level of an entire component, or individually for the multiple interfaces to that component.
- Adaptability. Components need to be built in a flexible way, so that they can be adapted and reused in different contexts. You can adapt a component at design or build time or even adapt at runtime.

- Development process. A development process that heavily utilizes components.
- Activities and roles. There are distinct activities for assembling from components – locate the parts, specialize them as appropriate, adapt between disparate views imposed by different parts that have to work together; versus building components
finding commonality in usage, re-factoring and generalizing the component with suitable parameters or « plug-points » for customization. Bottom-up approach. The availability of components and the feasibility of integrating them will likely drive time-boxed requirements as much as vice versa
- Early feasibility prototypes and tests. Partly related to the last point, it is essential to build early prototypes and conduct controlled experiments with individual and integrated components.
- Components need to be "composable". There must be an easy way of « gluing » them together, requiring minor enhancements or variations in use with current OO modeling languages and processes.
- Flexible approach of reflection. This mechanism provides a runtime representation of the components specifications that a programmer has written.

In component-based software development, software systems are built and configured using libraries of components. Applications can be adapted to changing requirements by reconfiguring components, adapting existing components, or introducing new ones.

Using of object-oriented programming languages and design techniques for development of component-based systems is not currently adapted for a number of reasons :

- Reusing aspects are introduced too late in development process : object-oriented analysis and design methods are domain-driven, which usually leads to designs based on domain objects and non-standard architectures. All these techniques are based on the assumption that applications are being built from scratch.

So reusing aspects of existing architectures, architecture styles and components in the development process are introduced too late.
- Lack of explicit components interactions visibility. Object oriented source code expose class hierarchies, but not object interactions. As a result, adapting an application to new requirements typically requires detailed study, even if the actual changes are minimal.

In order to solve these problems, composition environment and languages have to be based on an appropriate semantic foundation to understand all aspects of software components and their composition in terms of a small set of primitives and features.

4 Foundations Of Software Composition

Existing paradigms do not fully address the abstractions required for component-based development. A number of recent studies address the problem of discovering the right abstractions for software composition and definition of an unified paradigm which fulfil all composition aspects. To enhance adaptability and flexibility for component-based applications, we need to think not only in terms of *component*s, but also in terms of *architectures*, *scripts, coordination*, and *glue*.

A component is a «black-box » entity that both provides and *requires* services. These services can be seen as *« plugs »*. The main interests of components are due to the fact that the plugs must be standardized (i.e. a component must be designed to be composed [22]).

Components are elements of a component infrastructure; they adhere to a particular component architecture or « architectural style » that define the plugs, the connectors, and the corresponding composition rules. A connector is the wiring mechanism used to plug components together [30]. *Architectural Description Language* (ADL) may be used to specify and reason about architectural styles [30].

A script specifies how component are plugged together [28] . A scripting language allows to configure components, possibly defined *outside the language*.

Coordination mechanisms allow interactions between components considered as agents in a distributed (or at least concurrent) environment. A *coordination* language is concerned with managing dependencies between concurrent or distributed components. Classical *coordination* languages as Linda [22] and Darwin [25] may be applied.

Glue code overcomes the situations referred to as *compositional mismatches* [29] by adapting components to the new environment they are used in. *Glue* adapts not only interfaces, but also interactions contracts between the components or bridge platform dependencies. Glue code may be written to adapt a single component, or it may consist of generic abstractions to bridge various component infrastructures.

A precise semantics is essential to express foundations for software composition addressing multiple architectural styles and component models within a common, unifying framework. The simplest approach that seems appropriate is that of communicating, concurrent agents. Asynchronous *polyadic Pi-calculus* [26,23] is currently applied as a tool for modelling objects, components, and software composition. But the *tuple*-based communication of the *Pi-calculus* turns out to restrict extensibility and reuse. That leads many times to introduce communication of *forms* – a special notion of extensible records – instead of *tuples.*

Common approaches to formalize composition languages is the fact that all language features are defined by transformation to a core language that implements the *Pi L-calculus*, a *polymorphic* variant of the *Pi-calculus* [26,23], in which agents communicate by passing forms (specific extensible records). *Forms* and *polymorphic* extension are the major mechanisms for expressing extensibility, flexibility and robustness in the language.

5 Integrating The Composition Process In The Architecture Discipline

Figure 2 shows different architecture activities impacted by input information and producing new information or flows. Architecture is a transversal discipline involved in the following project disciplines :

- Analysis : the result is the specification of the architecture.
- Design : the result is the design of the architecture.
- Implementation and /or integration : the result is the assembling components.

Before starting any project, the strategy of the enterprise information system has to be clearly defined. The target architecture(s), and iterations to reach this target has to be approved by management and architects. According to the aims of these iterations, architecture will be defined in the scope of the iterations. It means that it will be specified in terms of refinement of business and *applicative* views of architecture. Functional and technical Needs, and requirements are expressed with Use case as input. Cartography models may be also used as an input. Project manager launches prototypes, defines project schedules and iterations, project and quality plan according to a risks analysis and according to architecture and strategy requirements. The design of architecture allows to clearly specify properties of the components and the services to invoke or reuse. Then, all these information allow to start composition activity.

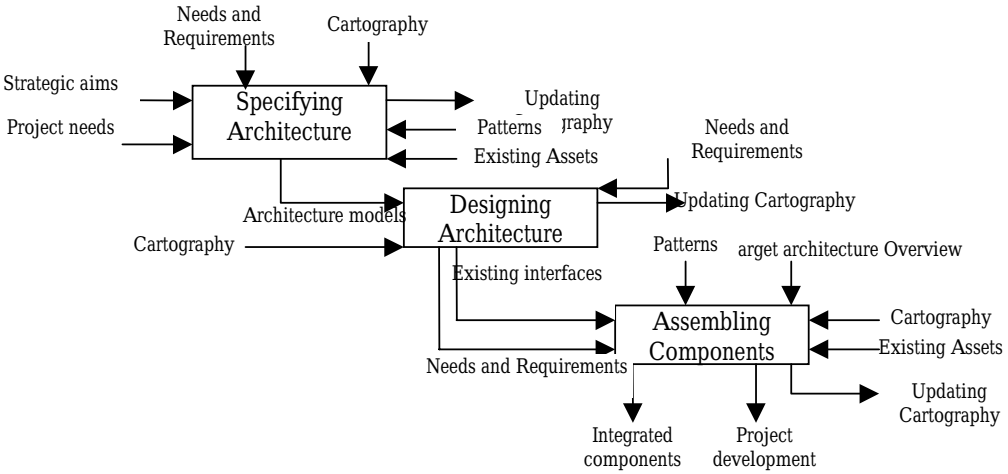

Fig. 2. : Composition as Architecture Discipline

6 Components Assembling Activities

6.1. Steps of Composition Process

Figure 3 is a simplified view of the three steps of this process.

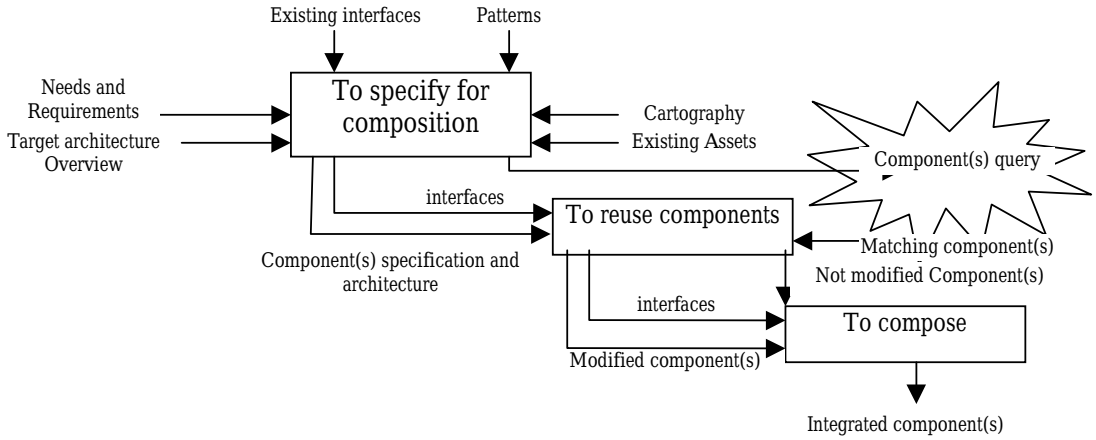

Fig. 3. : The Three steps of Composition Process

The components assembling process activity consists of the following three steps :

- Specification for composition purpose
- Reusing components
- Composing (reused component(s) or invoking remote services)

The specification of the component(s) is the cornerstone of this process. It is a refinement of the use case models done during the needs and requirements steps.

The *use case* which are used here are very low level description. Scenarios and sequence diagrams are used also. The specification allows to describe the needs in terms of functions, interface … that the required component has to fulfil. If this kind of component exists as legacy, or as a product to parameter (services plate form, ERP …), they may be directly reused without "*customisation*". Only the "glue" or *adaptor* has to be implemented else, customisation and glue have to be provided. Then, some adaptations have to be achieved concerning workflows, data repository …

6.2. Activities of Specification for composition step

The component identification stage takes as input the business concept models and the use case model from the upper activities as requirements discipline. Already made models from cartography may be used as input.

The goal is to identify an initial set of business interfaces for the business components and an initial set of system interfaces for the system components, and to pull these together into an initial component architecture. Any existing components or other software assets need to be taken into account too, as well as any architecture patterns you plan to use. In addition to identifying system interfaces, the identification stage also makes a first cut at the operations that need to be supported by the system. They are identified by name, but signatures and others details are added at a later stage. The system operations required are derived by examining the steps in the different use cases and deciding what the system's responsibilities are.

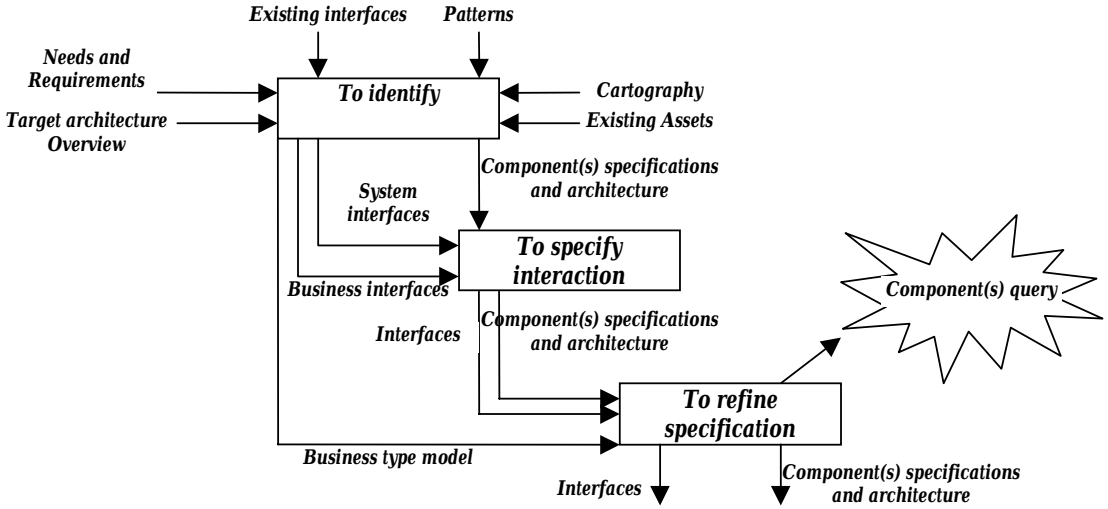

Fig. 4. : Specification activities for composition step

6.3. Activities of Reuse Step

If there is no matching component so the specified component has to be fully implemented and the process is the one of a classical project process. It means that all the disciplines have to be fulfilled. Else the component which has been bought or given (share ware) is analysed. Perhaps some documents about the component have been delivered. The component is compared to the requirements and a list of all the tasks to do and to schedule, to customize the component is done. Its interfaces may be modified, new functions may be added …
The implementation for customizing the component is followed by a test step, where *use cases* and requirements help for tests scenarios.

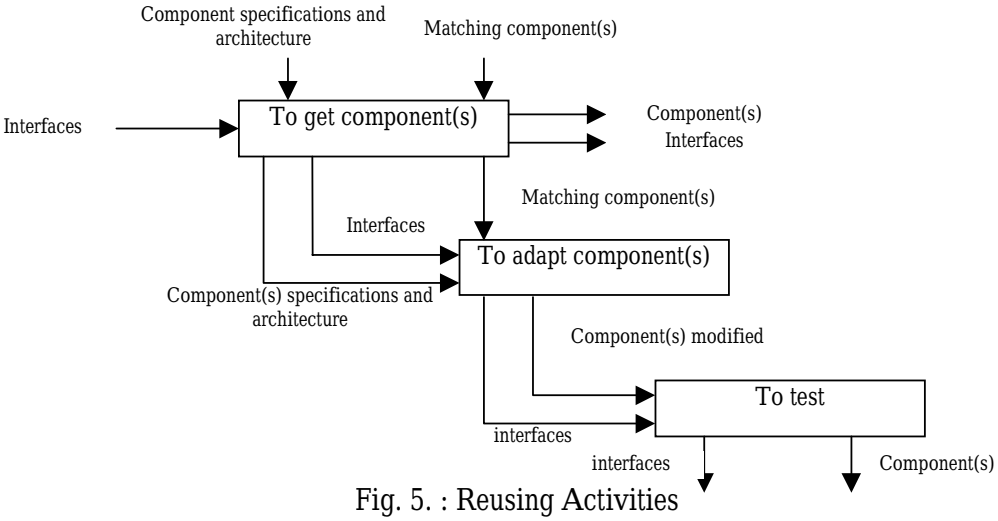

Fig. 5. : Reusing Activities

6.4. Activities of Assembling Step

The Assembling step ensures communication between the components according a business workflow. Adaptors are added statically or dynamically to the components for this purpose. Components repository is updated consequently. Specific workflows have to be revisited and to fit with the general and specific workflows. The workflows have to be coherent and then to be stored in a repository. Tests are then done including integration tests.

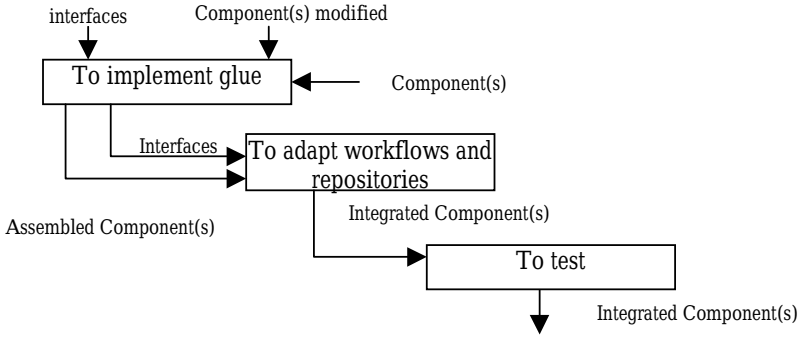

Fig. 6. : Assembling Activities

7 Conclusion

Although originally formulated for object oriented software development, the UP process has shown to be useful for composition. However, to provide adequate support for Component-Based Development, additional composition process has been added. In this paper, we have extended EUP with reusability and composition activities. In our opinion, our methodology is a pragmatic process for integrating object technology and component-based development. It may be applied to complement the unified software process for component-based development.We are working on these topics. This paper is only a part of a more global methodology we are consolidating by using it in different projects in the industry.

References

[1] Booch Grady et al. *The Unified Modeling Language User Guide*. Addison-Wesley, Reading MA, 1998

[2] Booch Grady et al. *The Unified Software Development Process*. Addison-Wesley, Reading MA, 1998.

[3] D'Souza, D.F. and A.C. Wills. Objects, Components and Frameworks with UML : The Catalysis Approach. Addison-Wesley, MA, 1999.

[4]Digre, T. «Business Object Component Architecture », IEEE Software, 15(5) : 60-69, 1998.

[5] Seacord, R.C. and K.C. Nwosu « Life Cycle Activity Areas for Component-Based Software Engineering Processes » TOOLS30 (eds. D. Firesmith, R. Riehle, G. Pour, and B. Meyer). IEEE Computer, Los Alamitos, CA, 1999, P. 537-541.

[7] Hansen, W.J. « A Generic Process and Terminology for Evaluating COTS Software » TOOLS30 (eds. D. Firesmith, R. Riehle, G. Pour, and B. Meyer). IEEE Computer, Los Alamitos, CA, 1999, P. 547-551

[8] Jacobson, I.; Bylund, S.; Jonsson, P., Using Contracts and Use Cases to Build Plugable Architectures, Journal of Object-Oriented Programming, May/June 1995.Rational Unified Process version 5.1

[9] Erich Gamma et al in *Design Patterns: Elements of Reusable Object-Oriented Software,*. (Addison-Wesley, 1995; ISBN: 0201633612):

[10] D. Garlan and M. Shaw, "An Introduction to Software Architecture," *Advances in Software Engineering and Knowledge Engineering,* Vol. 1, World Scientific Publishing Co., Singapore, 1993.

[11] G. Abowd, R. Allen, and D. Garlan, "Using Style to Understand Descriptions of Software Architecture," *ACM Software Eng. Notes,* Dec. 1993, pp. 9-20.

[12] Paul Clements, "From Domain Model to Architectures," A. Abd-Allah et al., eds., *Focused Workshop on Software Architecture,* 1994, pp. 404-420.

[15] Cornelius Ncube and al "PORE : procurement Oriented requirements engineering method for the component based systems engineering development paradigm http://www.sei.cmu.edu/cbs/icse99//

[16] Less Bass an al "Architecture Based Development" Technical Report CMU/SEI-99-TR-007, ESC-TR-99-007

[18] DSW Tansley and al « Knowledge based systems analysis and design » Prentice Hall 1993

[19] PMBOK guide from http://www.pmi.org

[20] Patrice Godefroid "Model Checking for programming languages using Verisoft" In Proceedings of the 24th ACM Symposium on Principles of Programming Languages, pages 174-186, Paris January 1997

[21] Nicolas Carriero and David Gelenter. *How to Write Parallel Programs : A Guide to the Perplexed*. ACM Computing Surveys, 21(3) : 323-357, September 1989

[22] Kohei Honda and Mario Tokoro. *An Object Calculus ofr Asynchronous Communication*. In Pierre America editor, Proceedings ECOOP'91, LNCS 512, pages 133-147. Springer, July 1991

[23] Markus Lumpe, Jean-Guy Scheneider, and Oscar Nierstrasz. Using Metaobjects to Model Concurrent Objects with PICT. In *Proceedings of Langages et Modèles à Objets'96*, pages 1-12, Leysin, Ocotober 1996

[24] Jeff Magee, Naranker Dulay and Jeff Kramer. *Structuring Parallel and Distributed Programs*. In *Proceedings of the International Workshop on Configurable Distributed Systems*, March 1992

[25] Robert Milner. *Functions as Processes*. In *Proceedings ICALP'90*, LNCS 443, pages 167-180. Springer, July 1990

[26] Oscar Nierstrasz and Laurent Dami. Component-Oriented Software Technology. In Oscar Nierstrasz and Dennis Tsichritzis, editors, *Object-Oriented Software Composition*, pages 3-28. Prentice Hall, 1995

[27] Oscar Nierstrasz, Dennis Tsichritzis, Vicki de Mey and Marc Stadelman. Objects + Scripts = Applications. In *Proceedings Esprit 1991 Conference*, pages 534-552. Dordrecht, NL, 1991. Kluwer Academic Publisher

[28] Johannes Sametinger. *Software Engineering with Reusable Components*. Springer 1997

[29] Mary Shaw and David Garlan. *Software Architecture : Perspectives on an Emerging Discipline*. Prentice Hall, April 1996